

VISIT...

LANZAROTE
Caliente.COM

GoLife I

A program that plays Go

**Not the way it should, but the way it likes
The concept of Quality in Computer Go**

Abstract

Computer Go is a hard AI problem, in some respects much harder than Computer Chess. The reason is not only due to a larger branching-factor, but also to the nature of the game; calculating the value of a position is more complex and time-consuming. This paper presents a program that relies heavily on a function called the Quality Function, a parameter-controlled function which serves as a model of the real-life knowledge of the game. The two major parameters are evaluated by performing a tournament on a set of instances of the program on a 9x9 board. The playing-strength is approximately 25 kyu.

**Henrik P Rydberg
Student MS Physics, Chalmers University of Technology
Gothenburg, Sweden**

Jan 1995

ACCKNOWLEDGEMENTS	5
THE GAME OF GO	6
THE RULES.....	6
SCORING	8
COMPUTER GO	9
MANY FACES OF Go	9
Go INTELLECT	9
GOLEM	9
MONTE CARLO Go	9
GOLIFE I - THE CONCEPTS	11
THE QUEST FOR THE QUALITY TREE	11
THE QUALITY FUNCTION	11
THE SUGGESTION OF MOVES.....	12
PROPERTIES, PROVERBS AND QUALITY	14
GOLIFE I - THE DEFINITIONS	17
THE QUALITY FUNCTION	17
APPROXIMATIONS - THE MODES	19
THE INFLUENCE FUNCTION	20
THE PROPERTIES - Q STATIC	22
THE PROPERTIES - Q DYNAMIC	26
GOLIFE I - BASIC STRUCTURES.....	27
THE POINTS.....	27
THE STRINGS.....	28
THE BOARD.....	29
GOLIFE I -THE ALGORITHMS	32
THE EYE ALGORITHM.....	32
THE BREADTH-FIRST SPATIAL SEARCH.....	34
THE ALPHA-BETA DEPTH SEARCH.....	34
GOLIFE I - THE SEARCH ENGINE	36
GOLIFE I - THE PARAMETERS	37
GOLIFE I - THE TOURNAMENT	40
LIFE	40
AJI.....	40
A SAMPLE GAME	42
CONCLUSIONS/FUTURE WORK.....	44
REFERENCES	45
APPENDIX	46

ACCKNOWLEDGEMENTS

I wish to thank Mats Nordahl for his supervision during the development of this program, and the many discussions on the topic. I also wish to thank Kristian Lindgren, who contributed with his knowledge of the game. There were many moments of joy and dispair watching the baby Golife grow...

THE GAME OF GO

Go is a game of pure skill; a strategy game like Chess, although different in many respects. It is one of the oldest games in the world: it was played in China at least 2500 years ago. In modern time the best players are still almost invariably from Japan, China and Korea, but the game has spread to the rest of the globe, and is increasing in popularity.

THE RULES

Go is played on a square grid made up of 9x9, 13x13 or 19x19 lines, upon which you place black and white stones, each player taking turn to place a stone. The stones are placed on the intersections of the grid, not in the squares. Once a stone is played, it never moves.

Stones can be *captured*, i.e., removed from the board, if they are completely surrounded by enemy stones. Only the points along the lines count as neighbors, i.e., the diagonal points are not considered as such. The free adjacent points to a stone are called the *liberties* of that stone; once a stone or a connected group of stones is out of liberties, it dies.

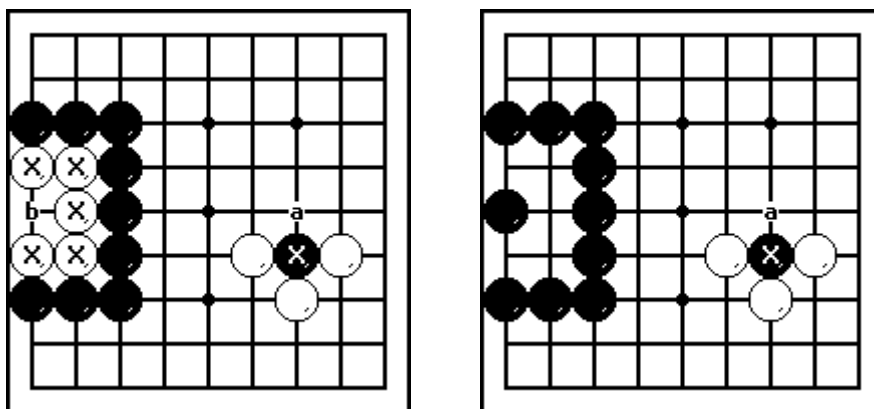


Fig. Both groups marked 'X' have one liberty each, 'a' and 'b' respectively. Black moves and kills the white group.

You are not allowed to place a stone at a point which has no liberties, unless you by doing so captures some enemy stones. After removing the killed stones, the placed stone will have liberties.

A group of stones is *alive*, i.e., impossible to kill, if it has *two eyes*. Eyes are surrounded points at which the opponent cannot place a stone unless he by placing the stone removes some of your stones in the process. If your group of strings has two such protected liberties, one liberty will always be in reserve if the opponent places a stone in the other, thus the group cannot be killed.

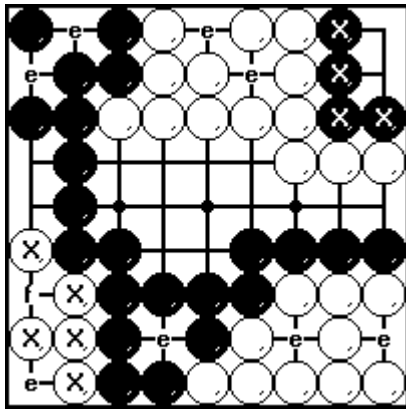


Fig. Live groups have two eyes. Real eyes are marked with 'e'. The groups marked 'X' are dead, because they cannot make two eyes. The white group at the left has one real eye and one *false* eye; it could be surrounded and killed.

Problems about whether a group can make two eyes are called *Life and Death* problems, and can sometimes be very hard to solve.

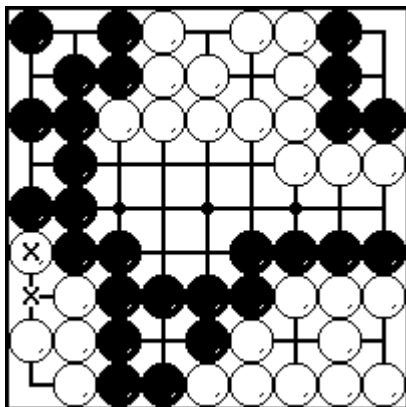


Fig. Ko fight. The marked stone can be captured again and again.

In the above situation, the capture of stones could go on forever; black places a stone at the last liberty of the white stone marked X; white kills the placed black stone by again placing a stone at A, and so on. To prevent this, there is a simple rule stating that any move leading to a position *exactly* the same as some time before in the game is forbidden; the same board can never occur twice. To solve the above situation, White has to play somewhere else first, Black has to play some where else, *then* Black can capture the white stone again. This kind of situation is called a Ko.

The goal in Go is to make the most *territory*. Territory points are such points from which the opponent can not make two eyes for his stones, which could be after a considerable number of moves.

During the game it is always profitable to place a stone, since more stones take control over more territory. But after a while when the board has been partly filled with stones, placing a stone might *take away* a territory point, a point that is already sure to be under your control. This move will not be profitable, and you can *pass* instead of placing a stone. When both players pass after each other, the game is over, and the one with the most territory wins.

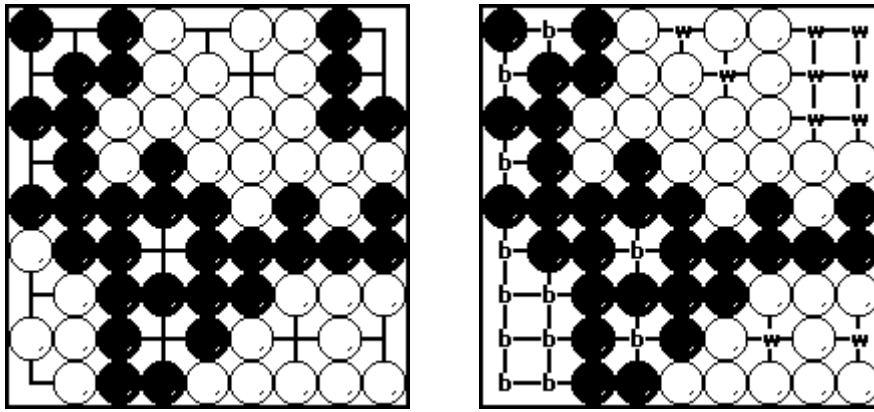


Fig. A finished game. The captured stones are kept and count as territory. Assuming no stones were captured before this state, black wins this game by $12+5$ to $10+4 = 3$ points.

SCORING

There are two different ways to count the territory: *Japanese* and *Chinese*. The two methods are equivalent in most situations, except in some very rare cases involving *seki* (Have not been able to find a good reference to this issue¹).

Japanese Counting

This is the most commonly used method: When stones are captured, they are kept as prisoners, and added to your total amount of territory after the game has ended. In this way, placing a stone inside your own territory will make you lose a point. Note that dead stones on the board count as twice the number of stones: First you remove them and count the territory, then you count them as prisoners.

Chinese Counting

A simple method of counting preferred in China: Captured stones are put back in the boxes, they don't count as prisoners. When the game has ended, all dead stones are removed, and both the stones and the territory count as points. In this way, the sum will always add up to the number of points on the board. Placing a stone in your territory does not change the score, and filling *dame*, the neutral points will gain points. The result, however, is the same as with Japanese counting. This is the preferred method to use in a computer program.

Ishi Press has many good books² for those who want to know more about the game.

COMPUTER GO

Although the rules of Go are very simple, it is yet one of the hardest games to master, not only for humans, but also for computers. In Chess, the best computer programs are now achieving master level of play, but in Go the best programs are still weaker than the average club player.

One of the problems with computer Go is the number of possible moves at each position, the *branching-factor*. In Chess this factor is 35 on average. In Go, the first few hundred (!) moves have a branching-factor of several hundred on a 19x19 board. Therefore, exact search on a whole board scale is out of the question. The only way to make exhaustive search achievable is to reduce the branching-factor. There are many techniques to reduce the tree in an exact fashion³, but still none is good enough to make full board search tractable. Inexact cutting of the tree must be done in a way such that the searched tree becomes an approximation of the full tree.

The search approach is the most common in the existing programs of today, but there are some who have tried different ways. Below is a list of the most commonly known programs, along with a few not so well-known programs which is listed here because of their unusual approach to the game.

MANY FACES OF GO

by David Fotland

This is a commercial program, and probably the most well-known. It has been under constant development for the last ten years or so. Fotland uses a knowledge-based⁴ approach to make his reduction of tree space. He has move-suggesting experts and a Joseki database. The playing strength is about 13 kyu. Fotland has also released a simpler version of his program: IGO, a version that only plays 9x9 Go.

GO INTELLECT

by Ken Cheng

This is the most serious competitor among the commercial programs, and it won the last Korea Computer Go Contest. It plays at about 13 kyu. The program is primarily built on knowledge about the game.

GOLEM

by Herbert D Enderton

Golem⁵ uses a Neural Network to reduce the game tree. Enderton further uses continuous values for group eyes, and eyes-space. The playing strength is about 14 kyu. This program shows that knowledge is very important in Go: Although the static evaluator uses no depth search, it can still play good moves.

MONTE CARLO GO

by Bernd Breugmann

This program uses still another approach: Simulated Annealing. Absolutely no *a priori* knowledge except the rules are used, and the program plays at 25 kyu on a 9x9 board⁶. The program simply plays a large number of full games to the end, where the evaluation is (relatively) simple. Based on such a simple algorithm, it is a remarkable program.

GOLIFE I - THE CONCEPTS

This document will try to cover the ideas behind the evaluation function of GoLife I, the program emerged from this project.

THE QUEST FOR THE QUALITY TREE

The first question to arise when making a Go program is this: *What constitutes a good position?* If a good answer could be found to this question, one approach to programming would be to make a min-max search, and simply make the move that leads to the best position.

Since the game-tree is so large, this full search - which indeed would be sufficient for any Go program that were run on a *very* fast machine - give rise to a second important question: *How do I select which moves to search?* Some kind of suggestions must be made inside the program, as to not cave in completely.

To answer the first question, the continuation might be something like this: *Could the good position be analyzed, divided into smaller parts, properties? What are those properties? How are they combined? What qualifies a property as a truly good one?*

In my program I have tried to find a general concept to be identified with the above questions, and that is simply Quality. The evaluation function used will be called the Quality function, a single-valued function which approximates the 'goodness' of a certain position on the Go board.

As a possible answer to the second question: *The action to select good moves to search is based on the Quality of the move.* With a perfect Quality function, no search would be needed, and in a somewhat less perfect function, one imaginably reasonable method of cutting the tree would be to throw away the bad ones; i.e. bad according to the same function.

Many researchers have expressed their belief that the one big difference between Chess programming and programming Go is the difficulty in finding a good evaluation function for Go - not so much due to the lack of interesting ideas as to the lack of fast machines; it would simply be too slow. Any good evaluation would have to consider the whole board and - which is worse - relations between stones on the board.

Now, with many good ideas about what makes a good position, they would combined constitute a fairly reasonable evaluation function. A slow one indeed.

So, my belief is that creating *approximations* to the ideas behind a rigorous and time-consuming property, or in some cases approximations to the definitions of the properties, would lead to an almost as good a function, but this time much quicker. Using these approximated functions to extract a subset of the moves, and other approximations to evaluate a search on this partial tree, is the method I have used in my program GoLife I.

THE QUALITY FUNCTION

Many people have through the centuries discussed and thought about Quality, but I find one of them especially within reach for people like me who have not studied philosophy: The ideas of *Pheadrus*, the insane man in the book *Zen and the art of motorcycle maintenance* by Robert M. Pirsig⁷.

The connection to Go, I believe will be this:

Suppose there is a genuine Quality of the game of Go, a value that can be assigned to every possible move. Call it Q. In the most general case, it will be a function of the board, the previous positions, the state of mind of the players, etc. Why the state of mind? In complex games like Go, situations occur where the knowledge about the game simply is not enough to tell what move is best. In such situations your might become aware of your opponents *style of play*, and try to make profit from that knowledge. However, in this report I have not made any attempts to interpret such concepts. Instead I will follow another path, influenced by Pheadrus:

Let Q be the genuine Quality function of n state variables:

$$Q = Q(\mathbf{s}) = Q(s_1, s_2, \dots, s_n)$$

Pheadrus talks about Static and Dynamic Quality. If we form the Static Quality from the state variables, and identify the Dynamic Quality with the derivate of Q, we can write down a recursion formula for Q:

$$Q^{n+1} = Q^n + Q_{Dynamic}^{n+1} \cdot (Q_{Static}^{n+1} - Q_{Static}^n)$$

The formula gives us the total Quality at step n+1, assuming we have knowledge about the Quality at step n. The Static and Dynamic Quality does not have to be scalars; they could be vectors or matrices, and the product could be a matrix product. As long as the basic concepts of Static versus Dynamic is used, the relation could be formed at will.

Now the properties can be divided into two categories: the vectors Qs and Qd. For example, the score of the board is a static quality. The influence on the board is a static quality. Taking sente is a pure dynamic quality. Playing far from strength is a dynamic quality. Playing Joseki moves is a dynamic quality; in short, the properties of the *board* is static quality, the properties of the *move* is dynamic quality.

So, the human values of the Go board can now be combined and interpreted in a relatively simple fashion, which makes it easier to see how the Quality function can reflect the game of Go.

THE SUGGESTION OF MOVES

The basic method of suggesting moves to play is to sort all possible moves using the Quality function, or some approximated instance of it. By cutting the sorted list, a cut in the search tree is made. This is necessary to make the problem tractable. It is also the most controlled way of cutting the tree; As to not make the search meaningless, we must assume that it is possible to *preserve* something when reducing searchspace. In full alpha-beta search, the main property is *completeness*. The speed is dependent on how we order our moves, i.e. on what predictions

we make. When cutting the tree, completeness is destroyed. There has to be something else that is preserved. That particular property is called *Quality*.

If I think I can kill some of my opponents stones, but this turns out not to be true, my search has preserved the *possibility*, and come to the correct conclusion that it is possible to kill the stones, *with less than perfect play*. This type of reasoning, built on a *result*, on the end node of the tree, would not lead to very good play, however, since going for something that turns out not to work often is not only a waste of time, but also disadvantageous in other respects. Suppose instead the search is made from the Quality measure. It is defined for *every move*, not just an evaluation of the endnode, such as "Can I" or "Can I Not" kill those stones. When cutting the tree, the Quality will diminish more gracefully, since "I like the moves I make". What could happen is that I fail to see moves that forces me to make worse moves than my opponent, but until that point, my moves have been impeccable, and the total error must be less than if I just evaluated the end node.

In GoLife I, only a subset of all moves are passed on to be sorted by the Quality function. This is solely due to time: In some cases, evaluating all the moves at one level and use a handful of them is too slow, and some moves will have to be discarded on the fly. By doing this, the Quality function is split into two parts: The *preprocessing* mechanism that tells what moves are outrageous, the *Evaluation* mechanism, a second thought. The relation between these two is not simple, and to be able to evaluate a certain definition of the Quality function, the preprocessing should be disabled. This is the main reason I have not implemented more sophisticated methods for move suggestion.

The most commonly used way of preprocessing the moves in existing Go programs is using pattern-matching, and this is indeed a nice concept: Humans make use of it all the time to make the same type of "first glance" pruning. My program will probably include such algorithms in the future.

PROPERTIES, PROVERBS AND QUALITY

In Go, there are many rules of thumb to help play the game. This is an attempt to analyze and place them in the correct category, as well as to discuss their conceptual definition. No attempt to make a complete list have been made, just list those that appear in the program.

<i>Property/Proverb</i>	<i>Qs</i>	<i>Qd</i>
<i>Influence</i>	x	
<i>Territory</i>	x	
<i>Value</i>	x	
<i>Life</i>	x	
<i>Aji</i>	x	
<i>Score</i>	x	
<i>Don't play close to thickness</i>		x
<i>Shape</i>		x
<i>Fuseki</i>		x
<i>Keep your stones connected</i>		x

As is indicated in the table, there is a strong relation between *Static Quality* and *Property* on the one hand, *Dynamic Quality* and *Proverb* on the other hand. The *proverbs* tells you in what way you should play.

In my program, GoLife I, I have chosen to combine the above properties into two two-dimensional vectors, according to the definition of the Quality function.

<i>Dynamic Quality</i>	<i>Static Quality</i>
<i>Don't play close to thickness, Fuseki</i>	<i>Value, Life</i>
<i>Keep your stones connected, Shape</i>	<i>Aji</i>

In this way, the move to make will be a function of two fundamental properties: *how big* a move is considered the Value, the Fuseki, the *feel* of the move, and *how stable* a move is considered the shape and the amount of aji it creates or destroys. The program will thus seek to make moves that claims large amounts of territory as well as attack and defend groups.

Influence

This is the basic static quality - it is commonly used in Go programs. It should reflect the ability to control points, the ability to survive attacks.

Territory

One suitable definition of territory is *a point at which the opponent can not profitably place a stone*. Placing a stone which will eventually die is not profitable. Thus the definition say that territory points are *controlled*. Dead stones that are already on the board will be territory since the point can be considered to have been territory when the stones was put. Neutral points count as territory under Chinese rules - the stones are worth one point - but not under Japanese rules; placing a stone at a *dame* accomplishes nothing. Territory is naturally derived from Influence.

Value

The value of a stone is an important property. It should reflect how much territory it makes, as well as how much territory it destroys. It should also reflect the *tension* of a certain position; a stone that makes only territory is less worth than a stone that both makes territory and destroys territory. This is a complex concept, and a precise definition will be given in the definition section of the report.

Life

The *life* of a group of stones should approximate the probability that the group will make two eyes. The category of Life&Death problems is indeed static, even if huge complex search would be necessary to find the correct answer; the game of Go is a finite state game, and as a such any Life&Death situation is determined. Natural dependencies for the *life* property is Liberties, Eyes, Influence. Search could be included as well, and is suitable for conditional updating.

Aji

Aji is a nice Go concept: the value of a group of stones, *regardless of whether it is alive or not*. This could be interpreted as a collective value of the opponents troubles; A stone inside the opponents territory creates aji *if* the opponent stones are threatened by that stone. This threat remains until the stone is killed or the threatened group is surely alive.

The opposite of *aji* is *bad aji*; it is a measure of the opportunities for the opponent to make my life miserable. If I make a safe move, the *bad aji* will increase from negative to something less negative. If I cut some enemy stones, the total *aji* will increase, i.e. my opponents *bad aji* will decrease. This is a very important concept in the game of Go.

Don't Play Close To Thickness

This is a typical dynamic property, an old proverb that tells you that you will be better off if you don't play too close to strong stones - neither the opponents nor yours. It can be derived from Influence.

Shape

This is a very valuable property. The reason it is considered a dynamic quality, when it indeed could be assigned to a position, is that it is not persistent; if you have done bad shape moves during the game, but still win the game by 1/2 point, it makes you a bad Go player, but it doesn't make you lose the game. You say 'ouuu' when you make a bad shape move, and it will be disadvantageous for a couple of moves, but then it is gone. The damage will then be measured in territory, not in bad shape. Shape could be composed from patterns, but also from simple properties like number of liberties and number of nearest neighbors.

Fuseki

The Fuseki, or *the beginning of the game*, it clearly Dynamic Quality. The Influence function could, if such a definition is made, favor the corners and sides before the rest of the board. The question is: *Why should influence with such a property be better?* I think the answer is: *Because it works better.* In other words, it is a disguised dynamic property. Experience tells us that we are better off if we start in the corners, and play certain patterns in the beginning. This property will depend on Space, the number of stones on the board, and similar properties.

Keep Your Stones Connected

This is a fundamental rule of thumb: Don't let the enemy cut your groups into many small groups, since sooner or later some of them will be killed. The knowledge is basically to value the *aji* property high, and it will appear in the program as a weight.

GOLIFE I - THE DEFINITIONS

GoLife I (GLI) has a very simple structure. It makes an N-ply adaptive search, and plays the best move found; no Joseki database, no Pattern Matching is used. Most of the time of this project has been spent on finding good properties to put in the Quality function, and simple ways to select which moves should be checked.

THE QUALITY FUNCTION

First, lets define the following variable:

$$M = \begin{cases} +1 & \text{If Black is to move} \\ -1 & \text{If white is to moves} \end{cases}$$

Let a *stone* take the value of 1 is the stone is black, and -1 if the stone is white. We define the *board functions* as

$$B(p) = \begin{cases} 1 & \text{If there is a black stone at } p \\ 0 & \text{Elsewhere} \end{cases}$$

$$W(p) = \begin{cases} 1 & \text{If there is a white stone at } p \\ 0 & \text{Elsewhere} \end{cases}$$

We need a metric to relate stones on the board. Let the *distance* between two stones a and b be defined as

$$d = |a - b|_{path} = \{ \text{The minimum number of steps needed to get from a to b via vacant intersections} \}$$

The steps are taken only along the lines, not along diagonals. Since only steps via vacant points are allowed, the distance between two stones totally separated by a line of opponent stones will be infinite. Note the similarity to *manhattan distance*, although the above definition refers to a *path-length*, and therefore is not a euclidian metric.

Let a *string* S be a solidly connected set of stones. Let an *m-group* be a collection of strings separated by a maximum distance m:

$$\begin{cases} G_{n+1} = G_n \cup \{S: \exists x \in S, y \in G_n: |x - y|_{path} \leq m\} \\ G_0 = S_0 \end{cases}$$

A group is thus grown from an initial string S. A 1-group is the same as a string.

Using the liberty of defining the relation between Static and Dynamic Quality at will, lets make the following definition:

$$\begin{cases} Q^{(n+1)} = Q^{(n)} + Q_{Dynamic}^{(n+1)} \cdot \Delta Q_{Static}^{(n+1)} \\ Q_{Static} = (Q_{Score}, Q_{Aji}) \\ Q_{Dynamic} = (Q_{Like}, Q_{Connect}) \end{cases}$$

The Static Quality is split into two parts: *Score* and *Aji*. The Dynamic Quality is split into the corresponding parts *Like* and *Connect*. The relation is set to be a scalar product, thus relating *Score* with *Like* and *Aji* with *Connect*.

The *Score* has the following form:

$$\begin{cases} Q_{Score} = \sum_S L(S)V(S) \\ L \in [-1,1] \\ V \in [-N, N] \end{cases}$$

Where the properties are defined as

L	Life of string; an estimate that the string will make two eyes.
V	Value of string; the amount of territory it contributes.

The *Aji* has a similar definition:

$$\begin{cases} Q_{Aji} = \sum_S (L(S) - 1)K(S) \\ K \in [-N, N] \end{cases}$$

Where

K	The value of killing the string, how much territory will be lost.
---	---

The *Like* variable is defined as

$$\begin{cases} Q_{Like} = \prod_i P_i \\ P_i \in [0,1] \end{cases}$$

Where each factor scales the total value, how much I like the move.

P	Proverbs; the knowledge that some moves are better than others, regardless of their worth in points.
---	--

The *Connect* parameter is just a constant, although it need not be.

So, the Quality function is now given in a load of variables, each depending on the situation or the move, and each being tied to a certain property of Quality which we can, or should be able to, describe as a continuous function.

APPROXIMATIONS - THE MODES

The Quality function has several levels of approximation, called the modes. The modes are defined below.

MODE_QUALITY

This is the mode used when updating the board after the computer or the opponent has made a move; all properties are calculated as in the original definition.

$$Q = Q + Q_{Dynamic} \cdot \Delta Q_{Static}$$

MODE_NORMAL

When looking ahead to find good moves, this is the mode used. The approximations are merely to limit the updating of the time consuming properties to an area around the placed stone. These properties include Influence, Life, Value, Aji.

$$Q = Q + Q_{Dynamic} \cdot \Delta Q_{Static}^{(normal)}$$

MODE_SPEED

When searching for combinations to cut and kill stones, no update of values are done, merely the groupings of strings and the life property. The updated properties include liberties, eyes.

$$Q = Q + Q_{Dynamic} \cdot \Delta Q_{Static}^{(speed)}$$

MODE_LIFE

This mode is used when looking for ways to make two eyes. The life parameter is only changed by this basic property.

$$Q = Q + (1, Q_{Connect}) \cdot \Delta Q_{Static}^{(life)}$$

MODE_LADDER

This is the fastest updating mode. Only connecting strings of different life parameter and liberties are properly set. This is used when following ladders to see if a rescue is possible.

$$Q = Q + (1, Q_{Connect}) \cdot \Delta Q_{Static}^{(ladder)}$$

THE INFLUENCE FUNCTION

Many of the properties defined above make use of a property called the Influence. This concept has been used for quite some time in Computer Go; most programs make use of something like it.

In GLI, the Influence is defined as

$$\left\{ \begin{array}{l} I(p') = \tanh(\beta F_B(p')) - \tanh(\beta F_W(p')) \\ F_B(p') = \sum_p B(p) D(C(p, p')) \\ F_W(p') = \sum_p W(p) D(C(p, p')) \\ D(n) \in [0,1] \\ C(p, p') \in [0, \infty] \end{array} \right.$$

where the variables are:

- D Decay map; the actual influence will diminish with path length.
- C Path length; the length of the path from p to p' through vacant points, measured in number of points. In this way the actual distance between two non-connected stones is infinite, i.e. the radiation is effectively shielded.
- β Steepness; controls the form of the influence function.

The $I(p)$ at stone points will not be affected of the above. The value at these points will always be $(B(p)-W(p))$. In this way, the Value of a board where all stones are 100% alive and every area on the board is sufficiently controlled (depend on β), the territory will simply be the sum of all $I(p)$; this is Chinese counting.

The most important property of the above definition is the shielding effect; whenever an area on the board is surrounded, all paths to that area is infinite, and the field from surrounding stones will be zero. Another property is the unstable condition of $\tanh(x)-\tanh(y)$ when x and y are large numbers: it is impossible to gain control over an area by increasing field strength, it must be done by shielding off the enemy stones. This is a real property of the Go game.

Note that this definition does not prefer any particular points on the board: it is not considered a static property to place stones in corners, unless that is a property of the field it self. In this definition, the field does not handle corners in a special way, and therefore no regard is taken to this fact.

The decay map could be any vanishing function. I have chosen the following form:

$$D_n(r) = \operatorname{arctanh}\left(\frac{1}{r^n}\right) = \frac{1}{2} \ln\left(\frac{r^n + 1}{r^n - 1}\right)$$

Which makes a single stone radiate as $1/r$ if $n=1$ and $\beta=1$. This choice of n is not entirely arbitrary; it seems to mirror the Go proverb N-step jump from a N-stone wall, which really tells a lot about the shape of the function in question.

Furthermore, the distance r is measured in 1-norm, *manhattan distance*. This is for convenience; r is simply the shortest path in gridpoints.

Modes:

RESET	$I(p')$ is reset and updated in a wide square about each stone.
NORMAL	$I(p')$ is updated in a small square around the placed stone.
SPEED	$I(p')$ is not updated.
LADDER	$I(p')$ is not updated

THE PROPERTIES - Q STATIC

All static properties in GLI have a basic definition, and a few approximate ones. A complete definition follows.

Value

The value is a property of string. The calculation is based on the influence field, which has to be updated first. The idea of V is to distribute the total sum of I(p) onto the strings on the board, in such a way that, at the end of the game

$$\sum_{S_i} V(S_i) = \sum_p I(p)$$

The criteria end of the game is important - any estimate will be sufficient during the game, but it should give the correct value when the game is over. To accomplish this, the influence I(p') of each liberty p' is distributed to several stones p according to:

$$\Delta_{p'}^{p'} = \frac{D(C(p, p'))}{1 + \tanh(\beta F_B(p')) \tanh(\beta F_W(p'))} \left(\frac{B(p)}{F_B(p')} \tanh(\beta F_B(p')) - \frac{W(p)}{F_W(p')} \tanh(\beta F_W(p')) \right)$$

The reason for this scheme is natural; When summed over all points p, we get the following expression for the contribution from point p' to Black and White respectively:

$$\Delta_B^{p'} = \frac{\tanh(\beta F_B(p'))}{1 + \tanh(\beta F_B(p')) \tanh(\beta F_W(p'))}$$

$$\Delta_W^{p'} = \frac{-\tanh(\beta F_W(p'))}{1 + \tanh(\beta F_B(p')) \tanh(\beta F_W(p'))}$$

When combined, i.e. when all groups that get contributions from point p' is alive, we get

$$\Delta_{B+W}^{p'} = \frac{I(p')}{1 + \tanh(\beta F_B(p')) \tanh(\beta F_W(p'))}$$

This form has the desired properties: for a completely surrounded point, one of the two tanh() in the denominator will be zero, and the contribution from that point will be just I(p'). At the end of a game, all strings form closed contours with the edges of the board, and all the points contribute as I(p'). The only exception is for dame, i.e. neutral points. These points will be surrounded by both kinds of stones. If the field is strong enough, tanh(βF) will be near unity, and the expression will be near zero. Thus the field parameter β controls the accuracy of the form.

The second important aspect of the chosen form is this: The value of a dead stone count exactly the same as the value of a live stone of the opposite kind, assume Chinese rules. This is

at least true at the end of the game, when dead stones inside territory certainly count as the opposite kind. To see this aspect of the definition, suppose all Black stones are alive and all White stones are dead, i.e. are assigned a life value of -1. We then get the following nice form for the contribution of point p' to the total value:

$$\Delta_{B-W}^{p'} = \tanh\left(\beta \left(F_B(p') + F_W(p')\right)\right)$$

This could indeed be interpreted as if the white stones were changed to black stones, as desired.

The strings now get their values according to

$$V(S) = \sum_{p \in S} \left(B(p) - W(p) + \sum_{p'} \Delta_{p'}^{p'} \right)$$

The signed unity B-W represents the value of the stones; each stone is worth one point, and this value is not shared with any other stones.

Modes:

RESET	V is reset and updated for every string on the board.
NORMAL	The change in outer influence is added to the value of the new string. The total V is still correct, but the distribution is somewhat distorted.
LIFE	The value M minus previous liberty value is added to the new string.
SPEED	The value M minus previous liberty value is added to the new string.
LADDER	The value M minus previous liberty value is added to the new string.

Bad Aji

The *killvalue* is a property of string. It is assumed that every point in a certain region around the string S will be lost if S is killed. This is a rough estimate, but it suits the purpose of not letting the opponent kill stones on the edge to a territory.

The influence I(p') of each liberty p' is distributed to the string S as

$$\Delta_S^{p'} = \frac{1}{|S|} \sum_{p \in S} \frac{B(p) \tanh(\beta F_B(p')) - W(p) \tanh(\beta F_W(p'))}{1 + \tanh(\beta F_B(p')) \tanh(\beta F_W(p'))}$$

The string S thus gets the total value of the surrounding points p'. The radius of surrounding points is arbitrary; and set to a fairly small value.

The total killvalue K is now simply

$$K(S) = \sum_{p' \in \text{Surrounding}(S)} \Delta_S^{p'}$$

The Bad Aji A is thus, as stated earlier

$$A(S) = (L(S) - 1)K(S)$$

Modes:

RESET	K is reset and updated for every string on the board.
NORMAL	The change in influence is added to K.
LIFE	The difference M minus previous liberty value is added.
SPEED	The difference M minus previous liberty value is added.
LADDER	The difference M minus previous liberty value is added.

Life

The purpose of the life property is to assign a value to the question how likely it is that a certain group will be on the board when the game ends. The most important factors in this property is the number of eyes of the group, the number of liberties of the group and the number of protected liberties, i.e. points that are controlled by the group.

In GLI, the Life property for a group is:

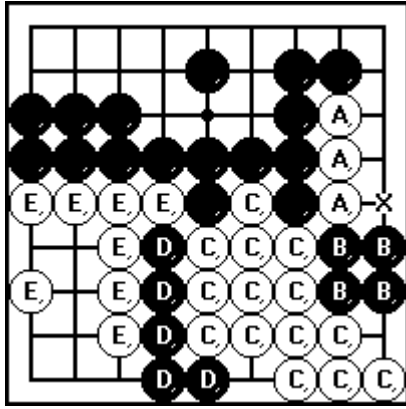
$$L(G) = \tanh(\sigma(P(G) - P_0))$$

where $P(G)$ is the number of protected liberties of the group, and σ , P_0 are constant parameters. Note that this definition works as a smooth stepfunction.

The function $P(G)$ is defined as

$$P_{n+1}(G) = P_n(G) + \max_{A \in \text{Adjacent}(G)} \left(N(G, A) \left(\sum_{B \in \text{Adjacent}(A)} P_n(B) - P_n(G) \right) \right)$$

where $N(G, A)$ is the probability that the opponent group A can be killed by string G . It thus measures the amount of extra liberties that is gained when string A is killed. The definition is as seen recursive; it mirrors the way groups are related in the following way: Suppose group A can kill group B , but the surrounding group C of B is assumed dead. Then killing group B will not be advantageous, since C and therefore A is dead anyway. Now suppose there is a group D surrounding C which could be killed by yet another group E , a group that is alive. Then $P(E)$ will transfer to group C in step one of the iteration, and to group A in the second. This loop will continue until no changes occur, or a predefined number of times.



White to move. Killing B by playing at 'x' is a necessary move.

The $N(G,A)$ function is at present defined simply as

$$N(G, A) = \begin{cases} 1 & \text{if number of liberties of } G \text{ is at least } \text{libs}(A) + 2 \\ \frac{1}{2} & \text{if number of liberties of } G \text{ is } \text{libs}(A) + 1 \\ 0 & \text{elsewhere} \end{cases}$$

Now, everything so far depend on the number of protected points $P(G)$ of an group G . So:

$$\begin{cases} P(G) = |F_m(G)| \\ F_{n+1}(G) = F_n(G) \cup \{x: (x \text{ is adjacent to } G \cup F_n(G)) \wedge (I(x) > T)\} \\ F_0(G) = \{ \} \end{cases}$$

The protected points are thus those in the vicinity of an m -group for which the influence $I(p)$ is stronger than some threshold T . With T slightly less than zero, this means all the liberties of all the strings in the group, plus some more points at a maximum distance of m from the strings in the group.

The definition in terms of groups have an important property: it handles connection between strings in a group. Since every string in a group will profit from the liberties of the other strings, it is indeed profitable to keep the stones connected whenever the threat on them becomes large enough. This is in line with the proverb *keep your stones connected*.

Modes:

RESET	L is reset and updated for every string on the board.
NORMAL	L is updated in a close range around the placed stone
SPEED	The aji of the killed stones are added to Qs.
LIFE	Direct connection and aji is updated.
LADDER	Direct connection and aji is updated.

THE PROPERTIES - Q DYNAMIC

Below is a list of the definitions of the dynamic properties accounted in the program.

Don't Play Close To Thickness

This property is calculated before the stones are put on the board. It is defined as

$$Th(p) = (1 - \alpha_r |I(p)|)$$

This will tend to spread the stones evenly on the board, and to take control over new areas, which is common practice in the *Fuseki*, the opening of the game. It will also have the effect that playing close to a strong line of stones will be disadvantageous; this is the proverb.

Shape

The *shape* is defined from the number of adjacent neighbors and diagonal neighbors connected to the new stone. This number is an index into an array of values. There are two arrays: one if the stones placed *connects* stones, and one if it does not. These arrays are typical parameters in the program.

Basically, the parameter works as an extra attenuation of these moves that cluster stones, in most cases considered bad shape. One example of the effect is the 'open triangle': When placing a stone in an open triangle, the new stone can see both the other stones, and it will know that they are both members of the same string. This will be bad shape. When placing a stone in a filled triangle, the diagonal stone will not be seen, and therefore it will not be as bad a shape. This is in line with Go knowledge.

Fuseki

This is defined by a matrix. This is a huge number of parameters into the program, but due to its simple usage inside, and the many symmetries of the Go board, it could be modeled by far less parameters which will hopefully be remotely complete.

I have used a FEM program to calculate this map as the solution to Laplace's equation with various boundary conditions. It is controlled by a handful of parameters.

Keep Your Stones Connected

This is yet another parameter to the program; it rates the importance of reducing *aji* in your own stones, and increasing it in your opponent's. This single parameter has a tremendous effect on the program's performance.

GOLIFE I - BASIC STRUCTURES

The basic structures of GLI, stripped of function definitions and private data, will be shown and explained. GLI is written in C++, and the language-specific details are assumed known to the reader.

The *board* is defined as a matrix of *points*, each containing information about the present influence field, and a flag that tells whether a point is a black stone, a white stone, or a free point. To be able to undo moves, all changed points are saved on a stack. Only those points that actually change will be saved, taking a lot less time and space than saving all points at all times.

THE POINTS

```
// =====  
// The Mirror structure, used for stacking boards  
  
struct Mirror {  
    char* mirror;  
    int bytes;  
};
```

The Mirror stores the position from which the data was copied, so that it can be restored later on without any external references.

```
// =====  
// The Point structure  
  
struct Point {  
    short stone;      // Flag:   1  Black stone  
                     //       0  Liberty  
                     //      -1  White stone  
  
    short ponuki;     // Whether lib is ponuki  
  
    int field[2];      // The field felt from stones in FIXUNITS  
  
    int infl[2];       // The resulting influence in FIXUNITS  
  
    int badaji;        // Bad aji of string in FIXUNITS  
  
    int lifeval;       // Life*Value in FIXUNITS  
  
    String* string;    // Base for string info  
  
    Point* next;       // Points are linked to strings  
  
    Mirror mirror;     // Used when saved  
  
    char fence[4];  
  
    // -----
```

```

    Point *env[4];    // The neighbor points

    Point *diag[4];   // The diagonal points

    int saved;        // Algorithmic aid

    ...
};

```

In addition to keeping track of the influence field and pattern of stones, there are two values represented here: **lifeval** and **badaji**. These two are the cornerstones of the program, and they are kept here to make an incremental update possible. Their definitions are given in the definition section. Whenever a point is a stone, the string will point to a structure containing information specific to that string. The stones in a string are contained in a linked list, therefore the next-pointer.

THE STRINGS

```

// =====
// The String structure

struct String {
    short libs;        // Number of Liberties

    short stones;      // Number of stones in string

    short ponuki;      // Number of adjacent ponukis

    short eyes;        // Number of real eyes

    int killval;       // Value of killing string in FIXUNITS

    int value;         // Value of string in FIXUNITS

    int life;          // Life of group in FIXUNITS

    int aji;           // Aji of string in FIXUNITS

    Point* head;       // First stone in string
    String* next;      // Linking of strings
    String* group;     // Grouping of strings

    // Group - specific information

    short geyes;       // Number of eyes of group

    short glibs;       // Number of libs of group

    short gprot;       // Number of m-th order protected libs of group

    short ginc;        // Amount of gprot gained if killed

    short gtot;        // Total amount of available gprot

    short dummy;       // Alignment

```

```

Mirror mirror; // Used when saved

char fence[4];

// -----

int saved; // Algorithm aid

...
};

```

The **libs** and **stones** properties are trivial. A **ponuki** is a single point, totally surrounded by your own stones. They are the basis of the real eyes, i.e. totally protected points. The algorithm for making **eyes** is somewhat less trivial; it will be given later on. The two properties **killval** and **value** have already been defined. So has **life** and **aji**; the latter is just the sum of *badaji* of the surrounding enemy groups.

The strings are gathered into *groups* as a linked list. The properties of the group are contained in the same structure for simplicity: Since every string in a group must be saved, the group structure can be changed without regard to reversibility. Also, letting every string in the group take the value of some group property, it will be valid even in modes where groups are not updated.

The **geyes** number is simply the maximum of eyes of all the strings in the group. The **glib** property differs from the sum of liberties of the strings, since strings can share liberties. This is a way to reflect the shape of the stones: Assuming that all strings have to be connected sooner or later, having a lot of shared liberties mean you effectively have less liberties. The **gprot**, **ginc**, and **gtot** variables are used to calculate the **life** parameter. The definitions was given earlier.

THE BOARD

```

// =====
// The Board.
//
// Black to play when atmove>0
// Legal moves are 0<=pos<SIZE, and pos==PASS, pos=TENUKI
// pos<=NULLMOVE is an illegal move

const int MODE_RESET=1; // No move, reset quality
const int MODE_CHECK=2; // Just check move validity
const int MODE_QUALITY=4; // Move, update quality function
const int MODE_NORMAL=8; // Move, Additive quality update
const int MODE_SPEED=16; // Move, Additive life update
const int MODE_LIFE=32; // Move, No quality update, Eye update
const int MODE_LADDER=64; // Move, No quality update, No eye update

// =====

class Board {
protected:
    int atmove; // Flag: +1 Black to play

```

```

//      -1   White to play

TagPos last[2];    // last[0] is last move by white
                  // last[1] is last move by black

Point* ko;         // Ko: This position is prohibited

Point* kostone;    // Ko: This stone is the ko stone

int koinfavor;     // Ko: If =atmove, Immediate recapture allowed

int captives[2];   // captives[0] is captured black stones
                  // captives[1] is captured whites stones

int komi;          // The komi

int qchange;       // The quality change of last move

int quality;       // The quality of board

Mirror mirror;     // Used when saved

char fence[4];

// -----

protected:
...
Point* pts_base;
...
(stack data etc.)

public:
int passvalue,life_libs;
float fieldgain,killgain,ajigain,lifegain;

float qd_thickhis,qd_thickmine,qd_ponuki;
float qd_connect_bulky[8],qd_bulky[8];

public:
...

Board(Param& param);
~Board();

void Reset();
int Move(int pos,int mode);
void Back(int mode);

...
};

// =====

```

The *board* structure holds the **point** matrix, as well as incremental values **qchange** and **quality**. It handles the stacking and update of moves, and contains a section of **parameters**, on which the Quality function is based.

The **ko** and **kostone** variables are special: they handle the simple case of cycles that appear in a game if a stone at a certain position is captured and recaptured over and over again: In Japanese rules, this is the only acquired test. If more delicate situations appear, such as a triple ko (gives a cycle after six moves), the players can agree on *No Result*, i.e. the game is considered never played. Computer programs, however, tend to get stuck if such situations occur. Also, in Chinese rules, the *No Result* does not apply. Therefore a correct test is made at the MODE_QUALITY level.

The **komi** is the extra points given to white in the beginning of the game. It is included in the **quality** variable.

GOLIFE I -THE ALGORITHMS

In this section some of the algorithms used in GLI will be outlined. The determination of Eyes, the Breadth-First search and Alpha-Beta search.

THE EYE ALGORITHM

The determination of the number of eyes of a collection of strings could be made in many ways. I have chosen to use a relation-based approach, i.e. one that does not look at shape, but solves a relation between strings in a graph. When looking for real eyes containing of points only, the group concept is somewhat different from the one defined earlier. Only strings connected via the diagonal connections in a *ponuki* will be part of the group.

An algorithm that finds out whether a group has at least two eyes, and therefore is perfectly alive, is

```
Function Alive(G)
{
  Let G be the group of strings connected via Ponukis only.
  Let P be the set of Ponukis in that set G
  Let S be any string in G
  Let p be any ponuki in P
  Let E(S) be the number of eyes of S, initialized to the number of ponukis of S
  Let change be true

  While change {
    change := false
    For all p in P {
      If there is an S adjacent to p such that E(S)<2 {
        change := true
        Remove p from P
        For all S adjacent to p {
          E(S) := E(S) - 1
        }
      }
    }
  }
  If there exist an S such that E(S)>=2 Return true
  Else Return false
}
```

Now, in GLI a slightly different algorithm is used, which gives a fundamentally different result; A measure of the *assumed* liveness is calculated, which means that the status can *change* as a consequence of the opponent making particular moves against the group. The advantage of this form is that the good eye-stealing moves can be directly spotted, another is that the good eye-making moves can be directly spotted too. A disadvantage is that there is no way of knowing when the group *really* has two eyes, i.e. when it is immortal. This could of course be solved by using the previous form, and is not a real problem.

```
Function TwoEyes(G)
{
  Let G be the group of strings connected via Ponukis only.
  Let P be the set of Ponukis in that set G
  Let S be any string in G
```

```

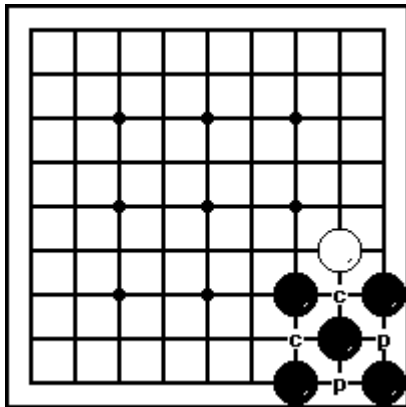
Let p be any ponuki in P
Let E(S) be the number of eyes of S, initialized to the number of ponukis of S
Let change be true

While change {
  change := false
  For all p in P {
    Let eyes := 0
    For all S adjacent to p {
      Let max := E(S)
      For all S' adjacent to p such that S' can connect to S through a point not in P {
        max := Max(max, E(S'))
      }
      eyes := eyes + max - 1
    }
    If max < 2 {
      change := true
      Remove p from P
      For all S in G adjacent to p {
        E(S) := eyes - number of shared ponukis still in P
      }
    }
  }
}
For all S {
  If S is adjacent to a p in P and E(S) >= 2 Return true
}
Return false
}

```

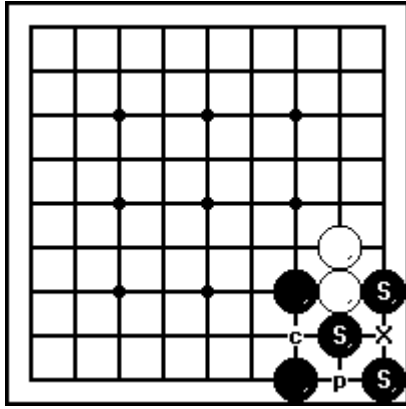
This algorithm makes two extra things:

It assumes that strings that can connect through non-ponukis (in one move, preferably) are one and the same, and therefore the maximum of those strings is used.



The p's are ponukis, and the c's are points at which the blacks stones can connect. This configuration therefore returns true.

It assumes that if a ponuki turns out to be worthless, it can be used to connect strings, and when this is done the total amount of presumed eyes left will be E-1 from each different string, minus the number of ponukis still assumed real shared between the different strings.



The point marked X is now a false eye, and in the first step of the iteration all the stones S will assume a value $(2-1)+(2-1)+(1-1)-1=1$, where the last one is from shared ponuki p. In the second iteration, all stones will assume the value $(1-1)+(1-1)+(1-1)=0$. Note that X has been removed from the eye set, and do not contribute (-1) as before. Also note that the upper right stone is unaffected by this last operation: that is why only strings adjacent to points in P is used to determine the number of eyes.

THE BREADTH-FIRST SPATIAL SEARCH

One very common activity in GLI is to scan the board for liberties and connections. To be able to do this efficiently and correctly, a breadth-first search is used, i.e. a search that starts with the liberties of a string, and then expands outwards from these liberties. The algorithm for this problem is well known, and is given below for reference.

Let S be a string

Let F be a set of free points, initialized to the empty set

Let Q be a queue of (points,distance)

Let p be a point

Let n be a distance

$Q := (S,1)$

$(p,n) := \text{GetFirst}(Q)$

While p is a valid point {

 If p is a liberty {

 If p is not in F {

 Add(F,p)

 If $n < \text{largest distance from string}$ {

 PutLast(Q,(p,n+1))

 }

 }

 }

}

This algorithm thus gives an ordered list F, where all the first-order liberties comes first, and thereafter second-order liberties, etc. Since F is a set, the points will only occur once.

THE ALPHA-BETA DEPTH SEARCH

This Algorithm was developed during the time when Chess programs were still weak, and the computers were slow as bricks...

The Exhaustive Min-Max Search is a fundamental algorithm in Game programming, and the Alpha-Beta search is an effective way to do this exhaustive search. Since exhaustive search is intractable, i.e. depend exponentially on the problems size, the best we can hope for is to reduce the *branching factor*, i.e. the average width of the tree as much as possible. This is what Alpha-Beta does. The trick is the following:

Let Q_α be the worst possible quality of the sequence to check

Let Q_β be the best possible quality of the sequence to check

Let $Q_{\max}$ be the best quality found so far

The algorithm's performance depend heavily on the ordering of the moves at each level; if the correct move is guessed at all times, it can be proved that the complexity C as a function of depth N is reduce from

$$C_{Exhaustive}(N) = B^N$$

to

$$C_{\alpha-\beta}(N) = \sqrt{B}^N = B^{N/2}$$

where B is the average branching factor. The algorithm thus ultimately doubles the number of ply's (depth levels) that could be searched.

GOLIFE I - THE SEARCH ENGINE

The search in GLI is divided into several depth intervals, or *modes*. At the first level, the moves are evaluated according to the mode `MODE_NORMAL`, which means all parameters are updated. When sorted, the best moves will be "big" moves and "severe" moves, i.e. moves that attack the opponent strongly.

In the following search, the "bigness" is dropped. Only the life parameters will be updated, but this allows for combinations to cut and kill stones.

The selection of moves goes like this:

```
Search(level)
{
  Let weak be defined by a tolerance T initialized to high tolerance
  While result is not good enough {
    Let W be the set of weak groups on the board
    For all strings S in W {
      Suggest and mark a set of moves from S (depend on weakness)
    }
    If at level zero and marked moves is less then required {
      Add free moves, i.e. Fuseki moves and generally big moves
    }
    Gather all marked moves, and Search next level. Set result of search.
  }
  Decrease T
}
```

The algorithm aims at making as little evaluation as possible, i.e. to make the first move that seems good enough according to the recursive scheme of min/max.

GOLIFE I - THE PARAMETERS

GoLife I is controlled by a set of parameters, contained in a single file. Below a typical configuration of such a file is given.

```
=====
----- GoLife I Parameter File -----
=====
```

== BOARD PARAMETERS

```
BOARD_SIZE      9
BOARD_MOVES     256
```

The board size and the maximum allowable depth of moves which can be saved in the search. Why save so many moves? Ladders, i.e. when a group is put in atari, the group is rescued, the group is put in atari again etc., tend to grow very deep indeed. This is not a complexity problem, since the branching factor is unity.

== FIELD PARAMETERS

```
FIELD_RANGE      5
FIELD_GAIN       1.6
FIELD_MAP         0.0000 3.8007 0.5490 0.3464 0.2553 0.2027 0.1682 0.1438 0.1256
```

The maximum range (manhattan distance) of a stone's influence field; the bigger the value, the more accurate the field *and* the more slow the Quality function. The **gain** parameter is used to scale the **map**, which gives the numerical values of the influence function. The values above are for an inverse distance function.

== STATIC PARAMETERS

```
LIFE_LIBS        10
LIFE_SIGMA        5

KILL_GAIN         1.0
AJI_GAIN          1.0
PASS_VALUE        0.4
```

These parameters are crucial to the Quality function; **life_libs** tell the zero intercept for the life parameter given in the definition section. **life_sigma** is the smoothness in ditto. The **kill_gain** value can be used to rate killing stones more or less than its actual value (according to the static quality definition). The **aji_gain** parameter scales the *bad aji* of a group.

Since GoLife I uses Chinese rules internally, it won't necessarily pass until almost all positions (except for two eyes) on the board have been filled by stones, since stones count as well free points. In Japanese rules stones don't count as territory, and placing stones in your territory would make you eventually lose the game. The **pass_value** can be used to make the program pass prematurely, whenever the size of a move decreases below that value. Normally any value below unity would be sufficient.

== DYNAMIC PARAMETERS -- PRE

QD_THICKMINE 0.0
QD_THICKHIS 0.0

QD_PREFERENCE
0.85 0.85 0.85 0.85 0.85 0.85 0.85 0.85 0.85
0.85 0.88 0.91 0.90 0.90 0.90 0.91 0.88 0.85
0.85 0.91 1.00 0.95 0.92 0.95 1.00 0.91 0.85
0.85 0.90 0.95 0.90 0.90 0.90 0.95 0.90 0.85
0.85 0.90 0.92 0.90 1.00 0.90 0.92 0.90 0.85
0.85 0.90 0.95 0.90 0.90 0.90 0.95 0.90 0.85
0.85 0.91 1.00 0.95 0.92 0.95 1.00 0.91 0.85
0.85 0.88 0.91 0.90 0.90 0.90 0.91 0.88 0.85
0.85 0.85 0.85 0.85 0.85 0.85 0.85 0.85 0.85

== DYNAMIC PARAMETERS -- POST

QD_PONUKE 1.3

QD_CONNECT_BULKY 1.0 0.95 0.9 0.8 0.7 0.7 0.7 0.7
QD_BULKY 1.0 0.8 0.7 0.6 0.5 0.5 0.5 0.5

The Dynamic parameters control the slope of change in the Quality function when making a move. The **thick** parameters are used to decrease the value of a move nearby **mine** or **his** stones. A value of zero disables the parameter, a value of unity makes some moves worth absolute zero. The **preference** map is a matrix that assigns certain weights to certain positions on the board. This is used as a scaling of the *score* Quality, and effectively makes the program play a comprehensive, although very simple, Fuseki. The map is constructed using a Finite Elements program, solving for Laplace's equation with certain boundaries. In this way, it only takes a few values to construct the map.

The division into *pre* and *post* is because some values apply *before* the stone has been placed, and some *after* it has been placed. The **ponuke** parameter is nice; it values a move that leads to a ponuke (star configuration) more than any other move of the same score value. Here one sees the neatness of the static/dynamic approach; making a ponuke is good, but not if the placed stone is worthless. If, however a stone is killed, and the result is a ponuke, it will (generally) become a very good move, which will also show in the Quality function.

The **bulky** vector parameters is for shape; they attenuate the value of a move that leads to bad shape, in GLI defined very simply in terms of how many of your own stones that can be seen from the stone you just placed. It will be different values depending on whether the placed stone will connect stones, or not. The vector values rate from one to eight such allied neighbors.

== SEARCH PARAMETERS

DEPTHSPEED 2
DEPTHLADDER 4
DEPTHCLEAN 6
WIDTH 16

TRIG_LIFE 0.95
TRIG_DEAD -1

TRIG_LIBS	3 4 4 4 4 5 5 5
RAND_VALUE	0.15

To control the search, the **speed**, **ladder**, **clean** and **width** parameters are used. They define the bounds of the search tree. The first three control the depthlevel of each type of search, the last the *minimum* number of moves to check at a certain level.

The types of search done is connected to the **trig_life**, **trig_dead** and **trig_libs** parameters. The first two mark an interval for the life parameter in which groups should be given special attention, i.e. that moves should be made from the stones of that group, aiming to defend or attack. The last vector parameter tells that strings of n (index) stones might be in trouble, suggesting that moves in the immediate vicinity of such strings might be necessary. The two types of attention is connected to different types of moves, and also works differently on different depth levels.

To make the program play "non-deterministic" in a certain position where it has many similar moves to choose from, the **rand_value** could be set to a fairly small value.

GOLIFE I - THE TOURNAMENT

To evaluate the two most important parameters, a tournament on a set of 8 instances have been performed. A *komi* of eight points have been used in all the games, i.e. eight points have been added to white to make the game even.

LIFE

In this tournament the (LIBS_LIBS,LIFE_SIGMA) parameters are varied. They model the number of points close to a 1-group of stones that has to be free for the program to estimate the group as alive. The estimate is a smooth function, where *libs* stands for the average number of free points needed, and sigma models the smoothness of that function; the larger the value, the smoother the function. The result from a full tournament is shown below. A positive value on a row means a win for the instance standing to the left in the table.

W \ B	(5,2)	(5,5)	(7,2)	(7,5)	(9,2)	(9,5)	(11,2)	(11,5)	#wins
(5,2)	X	13	4	3	-15	-44	-72	-22	3
(5,5)	17	X	-19	23	-1	-8	-23	14	3
(7,2)	-5	-27	X	-12	-9	-13	-11	-16	0
(7,5)	-3	32	-3	X	9	-3	-3	-16	2
(9,2)	19	-4	-8	-13	X	-19	-7	-8	1
(9,5)	-4	6	1	4	-13	X	-17	-11	3
(11,2)	-6	-69	-11	-7	-3	-22	X	-9	0
(11,5)	-62	-66	-62	-63	-61	-65	-62	X	0
#wins	5	4	5	4	6	7	7	6	

Two things are notable from this data: Firstly, the number of wins as black and as white differ the most for the instances that has high life rates, i.e. those that think they are easily alive. The last row actually says that the instance playing white has given up the game from the beginning, due to a tremendous over-estimation of black's initial territory (which is a function of how much alive a group is). It does tell though that being optimistic helps win the games, as long as the opponent does not try to punish the overplays, which is truly an important aspect to the real game. Secondly, the most stable instances have almost as many wins as black and white, and therefore can be considered to have estimated the true position a little better.

AJI

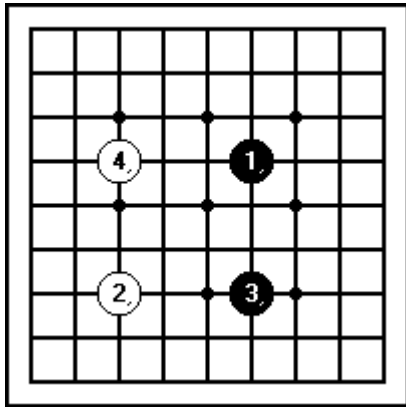
In this tournament, the AJI_GAIN parameter is varied. It controls how much attention the program should give to the correlation between groups and connections. It's a way to color the style of play so as to favor "cutting and killing" more as the *aji* parameter value gets higher. The *life* parameter is in these games set to (10,5), which seem to be where the program slightly overestimates it's own stability.

W\B	0.1	0.3	0.5	0.7	0.9	1.1	1.3	1.5	#wins
0.1	X	1	-18	-22	-22	-22	-22	-22	1
0.3	-7	X	10	-26	-4	-9	-8	-8	1
0.5	-13	4	X	-7	-7	-7	-7	-7	1
0.7	-42	4	-27	X	-21	-21	-21	-17	1
0.9	1	4	-21	-14	X	-9	-14	-14	2
1.1	1	-10	-52	-14	-14	X	-14	-14	1
1.3	-63	-65	-65	-63	-61	-63	X	-63	0
1.5	-63	-65	-65	-63	-61	-61	-63	X	0
#wins	5	3	6	7	7	7	7	7	

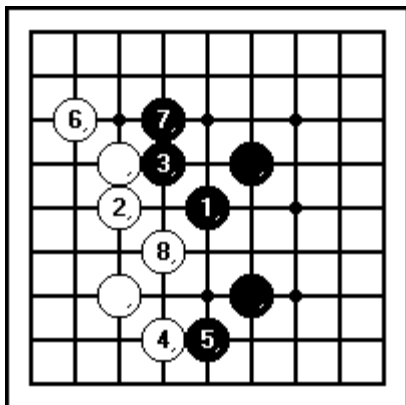
The tendency to win as black is very strong in this table: although the lower two lines does not tell much, the instances with a bigger *aji* parameter seem to do better than the ones with the smaller values. This then indicates the *very* important aspect of Go-playing: To attack and defend groups. The instances that fail to see the importance of that, just do not play as well.

A SAMPLE GAME

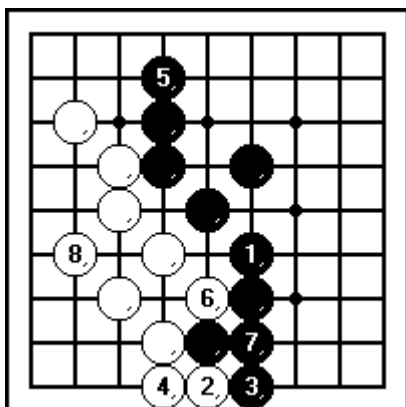
This sample game is played between two instances that differ by having (LIBS_LIBS,LIFE_SIGMA) set to (5,5) and (13,5) respectively. The (13,5) instance is playing black, and starts the game.



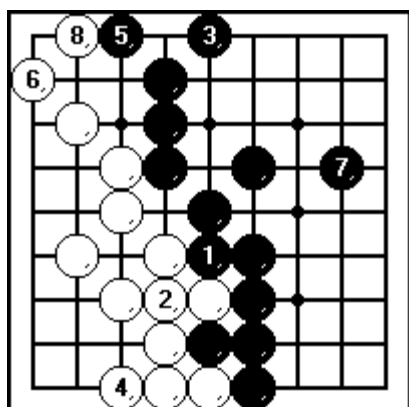
For a goplayer, this situation is a fairly easy game for black. Note how white has moved closer to the borders than black, in the hope that black wont manage to hold on to his initially bigger framework.



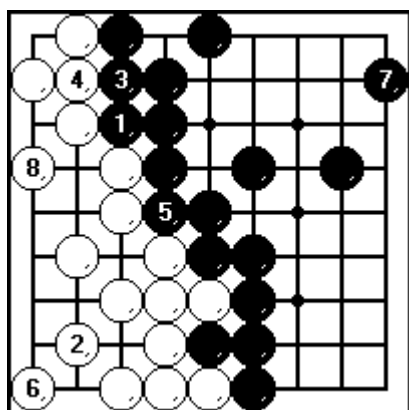
Although in a non-standard way (especially 6 should have been at 7), the two instances have managed to slide into the endgame, with a huge advantage for black.



The sequence 2-4 is very standard, but move 8 show the tendency for white not to trust it's own life sufficiently.



Only 5, 6 and 7 seem reasonable. Especially 7 firmly ends this game in blacks favor.



Only black take point's during this sequence, by filling the *dame* at 1,3 and 5. Next, the black program passes, but white keep placing stones until it only has two eyes left. The game ends with an 8 point win for black.

CONCLUSIONS/FUTURE WORK

In this project, I have concentrated on the fundamental principles of the game of Go, and I have found that good definitions of these fundamentals are important to the strength of the program.

Especially, the *life* and *aji* parameters serve as a fairly good model of how to value the moves. A single-ply estimate of the "bigness" of moves have proved to be sufficient for intermediate kyu-level play, but the *life* estimates must be used in deeper search.

Appreciating the importance of middle-game fighting, cutting and connecting is something all computer programs do rather poorly, mine is no exception. I therefore believe that a future development of a sound theoretical framework for these properties would increase the playing strength dramatically. My own strength is 4 kyu, and I know from experience that attacking groups and keeping my own healthy is the key to those playing strengths.

I plan to begin to work on *GoLife II* soon, and the major things I will redo and add is

- * The Move engine of the simpler modes is fast, but the more complex needs to get a lot faster
- * Invoke Search in the estimate of the *life* parameter; It should keep track of both Life of groups, and Ladder-Life of separate strings. Also, these states should be updated sparsely, i.e. the dependent positions should be saved along with the status.
- * Use a pattern library to improve the speed of the move suggestor.

As I have already pointed out, I believe that a better understanding of the fundamentals from a program's point of view is needed, but that is a totally different project, and a mighty big one.

REFERENCES

- [1] NewsGroup rec.games.go, nov 94 - jan 95
- [2] Ishi Press, Author, *In the Beginning*; James Davies, *Life and Death*; Author, *Attack and Defense*
- [3] computer-go mailing list, 1 jun 1994, David Dyer, *Notes on Searches, tree pruning and ordering in Go*
- [4] ftp, 27 feb 1993, David Fotland, *Knowledge Representation in The Many Faces of Go*
- [5] School of Computer Science, dec 1991, Herbert D Enderton, *The Golem Go Program*
- [6] Physics department, Syracuse University, 29 march 1993, *Monte Carlo Go*
- [7] Robert M. Pirsig, *Zen and the art of motorcycle maintenance*

APPENDIX